JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. 2000; **12**:121–141

Practice

Pattern-based software reengineering: a case study

William C. Chu[1,*,§], Chih-Wei Lu[2], Chih-Peng Shiu[2] and Xudong He[3]

[1]*Department of Computer and Information Science, TungHai University, Taichung, Taiwan, ROC*
[2]*Department of Information Engineering, Feng Chia University, Taichung, Taiwan, ROC*
[3]*School of Computer Science, Florida International University, Miami FL, U.S.A.*

SUMMARY

Most legacy software systems were developed in imperative languages with traditional design approaches. Instead of continually maintaining these legacy systems in their original architecture and design at high cost, reengineering them to new systems with good design and architecture can significantly improve their understandability, reusability and maintainability. Design patterns (DPs) combine successful established design practices and experts' experiences into a set of inter-related components that exhibit known behaviours with better flexible structures. Software development with DPs provides easier understanding and standardization that make system evolution much more effective. In this paper, we use a parallel program generation environment (PPGE) as a case study to demonstrate the reengineering of a traditional software system into a pattern-based software system. An architecture using the dynamic-packing component library (ADPCL) composed of existing well-known design patterns, and a pattern-based reengineering approach for the transformation of systems are proposed. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: legacy systems; software architecture; design patterns; software reuse; software components; dynamic packing

1. INTRODUCTION

Most legacy software systems were developed in imperative languages with traditional design approaches. The coupling relationships of these legacy systems are high and their functional/data dependencies are complicated. As a result, the maintenance cost of these traditional systems has been

*Correspondence to: Dr. William C. Chu, Department of Computer and Information Science, TungHai University, Taichung, Taiwan 407, ROC.
†E-mail: chu@cis.thu.edu.tw

Contract/grant sponsor: National Science Council of Taiwan contract; contract/grant number: NSC89–2213–E–029–004

Received 11 August 1999
Revised 21 January 2000

extremely high compared to that of the initial development cost [1]. Software maintenance and software evolution involve the identification or discovery of program requirements and/or design specifications that can aid in understanding, modifying, and adding features to the old programs. The tasks of software reengineering include the recovery and the recording of high-level information about the system structure, functionality, dynamic behaviour and design rationale. Besides, software reengineering of an existing system produces a better model and a system with improved maintainability.

Design patterns (DPs) [2] have integrated successful established design practices and the experiences of experts into a set of components that exhibit known behaviours with better structures. DPs are considered to be one of the most forward-looking methods for modern system analysis and design [3,4]. They provide not only a common base for better communication among software personnel, but also increase the reusability and productivity for both forward engineering and reengineering. Instead of continually maintaining legacy systems in their original architecture and design at high cost, reengineering them into new systems with good design and architecture can significantly improve their understandability, reusability and maintainability.

In order to experiment with design-pattern based reengineering, we selected the parallel program generation environment (PPGE) [5] as the experiment system. Programming on distributed memory parallel machines involves tremendous effort on the part of programmers. Programmers need to consider the architecture of a parallel machine, load balancing, data communication among the processors, parallel algorithms, and many other issues. Without tool support, parallel programming is labour intensive, error-prone and tedious. The PPGE was designed to offer a semi-automatic environment for the generation of parallel programs for solving partial differential equations (PDEs) on distributed memory computing environments, especially for clusters of workstations and parallel machines with wormhole-routed interconnection networks. The PPGE consists of the graphical user interface (GUI) package, the algorithm package, and a set of different parallel computation models. However, it has a high maintenance cost and is difficult to extend with third party solutions and algorithms. Furthermore, it lacks a run-time reconfiguration capability, which is necessary to optimize the parallel solution. Moreover, the PPGE is a standalone system, not a web-based system, which keeps the PPGE from offering services to multiple users and multiple domains. Reengineering the PPGE into a system with the above missing features and a better model could greatly improve the process of parallel programming.

In this paper, an *architecture with dynamic-packing component library* (ADPCL), which is composed of many existing well-known DPs, was proposed as a major solution component of this reengineering case study. The unified modelling language (UML) was used to represent the reengineered model, so that a common notation can be reused for future maintenance and evolution. ADPCL exhibits the expected features of high reusability, high extensibility, easy evolution, and a design-pattern based architecture. The JAVA language was the target language for the reengineered system, so that the resulting system would have favourable interoperability characteristics.

From this case study, we have discovered the advantages of applying DPs to the reengineering process and have prepared this paper to share the experience and results with other researchers and practitioners. This paper is organized as follows. In the next section, a survey of important related works is provided. Section 3 describes the overview of the features and the architecture of ADPCL, along with the approach and implementation of reengineering PPGE into a pattern-based system. Section 4 discusses some experiences and evaluations of this case study. Finally, Section 5 concludes the paper with a summary of some of our experiences and discussions of possible future work.

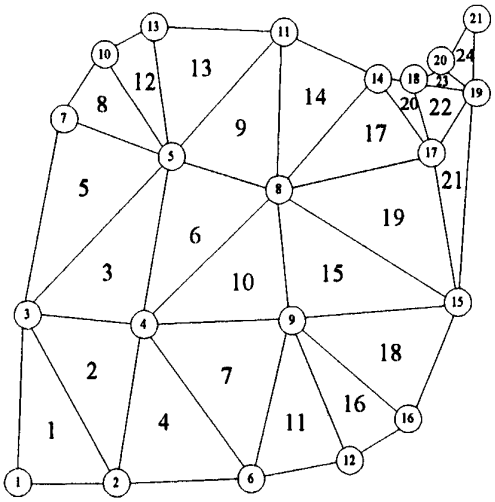


Figure 1. A finite element graph.

2. RELATED WORK

2.1. Overview

Software reengineering involves understanding both the source system, which is the PPGE in this case study, and the target system, which is a pattern-based PPGE. In this section, we first give a brief overview of the finite element model (FEM), which is the underlying model of the PPGE. Then, we discuss selected related technologies for the target system, including object-oriented (OO) technologies and design patterns. Finally, we describe the related reengineering approaches that are used in the migration from a source system to a target system.

2.2. The parallel problem modelling in FEM

The finite element model (FEM) has frequently been used to solve parallel problems [6]. In the FEM, a physical object can be represented as a Finite Element Graph (FEG), which is a connected and undirected graph that consists of a number of finite elements. Figure 1 shows an observed object and its corresponding FEG with 21 nodes and 24 finite elements.

Each finite element is composed of a number of nodes. The number of nodes is determined by the application. In the context of parallelizing programs using the FEM with iterative techniques [7,8], a parallel program may be viewed as a collection of tasks represented by the nodes of an FEG. Each node represents a particular amount of computation and can be executed independently. To execute efficiently an FEM program on a distributed memory multi-computer, the program needs to

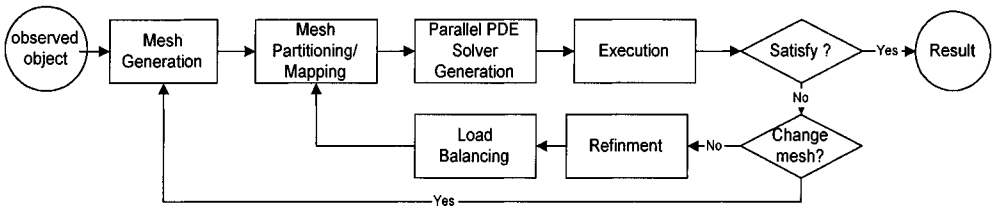


Figure 2. A typical process of parallel programming using FEM.

distribute the nodes of an FEG to the processors of a distributed memory multi-computer as evenly as possible, and minimize the communication cost of processors. Since finding the optimal solution of this mapping problem is known to be one of NP-completeness [9], many heuristic methods were proposed to find satisfactory sub-optimal solutions. Some of the work focused on better algorithms for specific problems, for example [10–15]. Other work involved constructing environments or frameworks, for example [16–19].

Figure 2 shows a typical process of parallel programming using the FEM. It consists of the steps of mesh generation, mesh partition/mapping, PDE solver generation, execution, mesh refinement and load balancing. The whole process may be applied iteratively using a different number of processors, mesh, partitioning, solver, refinement or load balancing algorithms. The execution results are usually collected and analyzed to achieve a desired performance. Obviously, this process is very tedious and costly.

In this process, different methods and algorithms may be adopted in each phase to find an acceptable solution. Solving parallel problems is so irksome because finding an acceptable solution takes iterative and tedious efforts. The PPGE has been designed to integrate most of the available methods/algorithms and to provide tool support for the whole process. The PPGE is a complicated software system that consists of reusable process, algorithms, and a set of tools that are required to be maintainable and reusable. However, the PPGE is currently very difficult to maintain and reuse due to the tightly coupled components. In order to make maintenance and evolution much more cost effective, the PPGE needs to be reengineered to a better style and architecture.

2.3. OO technologies and design patterns

Object-oriented (OO) technologies greatly influence software development and maintenance on the aspects of cost saving and quality improvement [20–22]. Our objective in reengineering the PPGE using OO technologies was to increase its reusability, maintainability, scalability and portability. Design patterns provide higher level abstraction than fragmented and procedure-based reuse approaches. The basic idea behind design patterns is that common idioms are found repeatedly in software design and these patterns should be made explicit, codified, and applied appropriately to similar problems. The gang of four's book [2] classifies design patterns into three categories. *Behavioral* patterns concern algorithms and the assignment of responsibilities between objects.

Structural patterns deal with how classes and objects/functions are composed to form larger structures and functions. *Creational* patterns centre on the instantiation process of how objects are created, composed and represented. Solving PDEs on distributed memory computing environments for different problems is complicated but similar. Hence, it is appropriate to apply design patterns to reengineer the PPGE.

2.4. Software maintenance and reengineering

Software maintenance is the process of changing a system in order to correct or enhance the software after its delivery [23]. Maintaining software is not only error-prone but also very costly. Software reengineering is concerned with re-implementing software systems to be more maintainable [23,24]. Pleszkoch, Linger and Hevner [25] focused on the feasibility and techniques of transforming unstructured source code to structured code. Gall *et al.* [26] presented a capsule oriented reverse engineering method (COREM) for transforming a traditional imperative program into an OO architecture. Gall, Kolsch and Mittermeir [27] proposed applying patterns in reengineering. The major objective was to improve the reusability and maintainability of the system and the efficiency of the reverse/reengineering process. The focal point of the above work is the source code. The experiment presented in this paper is mainly based on design documents and front-end expertise, and only partially on source code.

Gleich and Kohler [28] introduced a reengineering tool, FAMOOS, for object-oriented systems, along with tools implemented using ADA83 to aid the process of the analysis of the system architecture. In order to preserve the integrity of reengineered OO systems, testing mechanisms like built-in tests (BIT) [29] have been proposed for object and class hierarchy self-testing, and for testing software reuse and maintenance. During the process of reengineering or forward engineering, Jahnke and Zundorf [30] tried to distinguish poor and good DPs using generic fuzzy reasoning nets (GFRNs), and provided some recommendation for improvement.

3. REENGINEERING PPGE INTO A PATTERN-BASED SYSTEM

3.1. Goals and steps

The four goals of our reengineering approach were to introduce the following features into the PPGE.

- The scalability and flexibility of adding/adopting new algorithms to the PPGE without affecting most parts of the PPGE.
- The dynamic reconfiguration of software components—dynamically binding/linking alternative algorithms in each phase of parallel programming without re-compiling the whole program.
- The maintainability and reusability—making the PPGE much easier to evolve with new technologies.
- A web-based open system for the developers and users worldwide.

The five steps of our reengineering process are as follows.

- Recover the system architecture and object model from a legacy system. This recovery may involve collecting information from documents, domain experts and source code, so reverse engineering may be required in this step.
- Define a target framework or architecture with good patterns to be reengineered into the legacy system.
- Decompose, restructure and classify components in the legacy system into behaviour, *structure* and *creation* classes according to the definitions in [2].
- Map function/object components in the object model of the legacy system into patterns.
- Reengineer components in the legacy system into patterns.

3.2. Design recovery of the PPGE

The design recovery of the PPGE was mainly based on the design documentation and related papers, knowledge from the designers, and the source code. In this phase, the PPGE is first decomposed into subsystems, then the behaviour, architecture, and properties of each subsystem are reverse engineered and recorded. Our pattern-based reverse engineering is *pattern-oriented*, which mainly focuses on the recovery of DP-related information. The guidelines of DP-related information are easily available in [2]. The majority of the PPGE information was collected from published papers [5] or technical reports, and incomplete information was revealed from source code. The more DP-related information that can be collected, the easier the reengineering process is. In this case study our implementation was mainly gathered manually.

The design recovery yielded nine major components in the PPGE, including a coordinate model generator, a mesh generator, a mesh partitioner/mapper, a PDE solver generator, an executor, a mesh refiner, a load balancer, a performance analyzer and a GUI facility. The recovered system architecture of the PPGE is shown in Figure 3.

The *coordinate model generator* is responsible for generating a predefined format of the PPGE coordinate model for the observed object. In the PPGE, a 2D/3D coordinate model of the observed object can be obtained in any of the three following ways: (i) manual input of coordinates; (ii) a transform coordinate model from systems such as DIME, Metis, Jostle and Party; and (iii) output from a digitizer or image processing. The *mesh generator* then accepts the coordinate model as input, meshes this model, and generates the corresponding finite element graph (FEG). After the corresponding FEG is generated, the display module, the *FEG drawer*, can display the FEG graphically. The *mesh partitioner/mapper* partitions the FEG and maps the partitioned FEG to processors according to the selected algorithm. The partitioned FEG can be displayed in different colours graphically in the PPGE, where nodes mapped to the same processor are drawn in the same colour. The PDE *solver generator* contains a set of the PDE solvers, and is responsible for generating the corresponding C+MPI or Fortran+MPI code of the selected solver.

The task of the *executor* is to compile the generated solver program and to load it automatically to distributed parallel machines for execution. Since our solver program is written in C+MPI or Fortran+MPI, the PPGE can use either MPICH [31], LAM [32], WinMPI [33], or the high-performance communication library (HCL) compiler to compile the solver program. Currently, the major testbeds of the prototype PPGE are on workstation clusters, PC clusters or an IBM SP-2. The execution is based on the SPMD (single process multitude data) model. Since the results from solving the input FEG may not always be acceptable, the *refiner* is used to further refine the input FEG. After the refinement, the

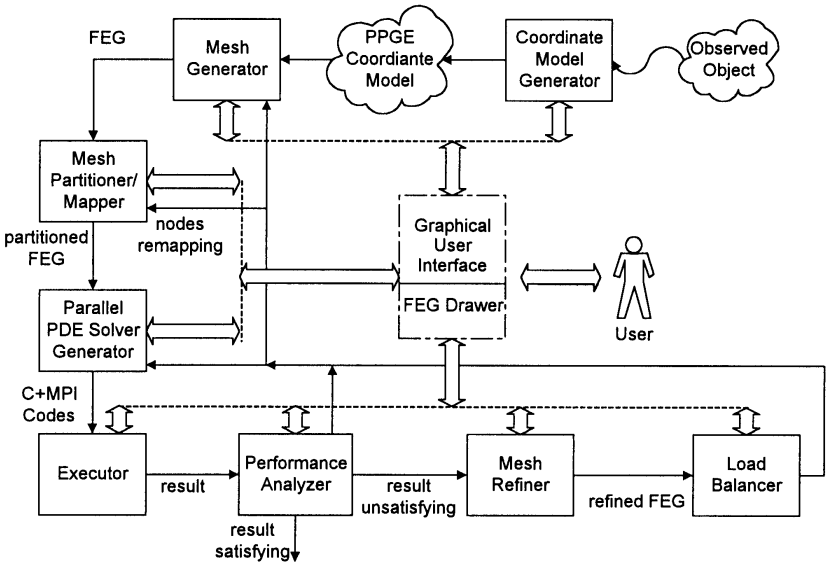


Figure 3. The system architecture of the PPGE obtained from design recovery.

computational load of each processor may become unbalanced. The *load balancer* is responsible for re-balancing the computational load of each processor and minimizing the communication cost among the processors. The steps are repeated until the desired results are achieved. The selected algorithm in each phase and the execution result is recorded in the PPGE. The *performance analysis generator* gathers the execution results of selected algorithms and the number of processors, which include the details of the computation and communication time statistics. The *user interface facility* provides the graphical user interface to the user. Most of the interactions between the PPGE and users are the selection of algorithms from pop-up menus.

3.3. Target architecture—ADPCL

In this experiment, a three-tier architecture, ADPCL (*architecture with dynamic-packing component library*) is introduced. The ADPCL exhibits the features of high reusability, easy evolution, and a design-pattern based architecture. It is aimed at offering a software architecture to the application domain like Unix `pipe`, in which input data are transformed into desired output data through a sequence of operations. Many applications are very easily modelled as pipe applications, including transformation systems like compilers, parallel program generation systems or certain web-based programs.

The ADPCL contains three major parts: *application process management*, a *component service agent*, and a *substitutable component parts library*. Figure 4 shows the system architecture of the ADPCL. This specific architecture intends to isolate the application process and data so that the

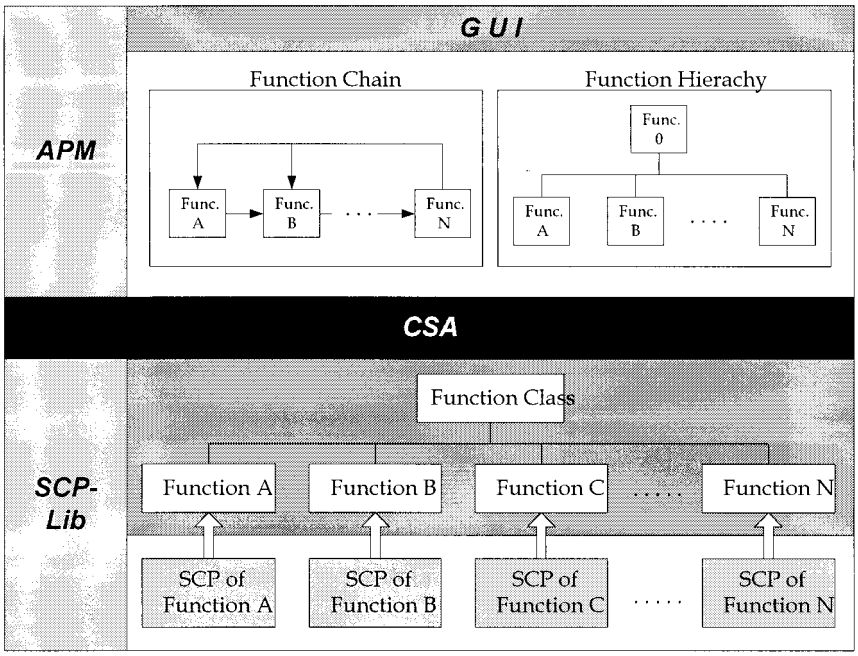


Figure 4. The system architecture of ADPCL.

components have low coupling and dependency between one another. This architecture also tends to accomplish the dynamic reconfiguration of software components.

The *application process management* (APM) part contains a GUI framework and a general application framework. The APM clearly sets up a boundary between the GUI components and the application components so that the implementation of the GUI components and the application components do not depend on each other. The old implementation of the PPGE mixed these two sets of components together. The general application framework is specifically designed for software systems that behave like pipe processes, such as transformational systems. The general application framework also offers the flexibility of selecting alternative services in each phase of the function chain. As a result, the APM can easily support roll-back optimization by feedback from other phases of the process and/or the flexibility of selecting appropriate alternative solutions in each phase.

Another design facility of the ADPCL is to set a clear boundary between processes and resources. Resources are data or functional objects. The *component service agent* (CS-agent) serves as an agent to provide services to meet APM requests, so that the requests from applications (clients) do not directly affect the components that provide the services. The CS-agent also conceals the details of the composition and location of the services from the client components. As a result, it is easy to alter communication features by changing the protocol or migrating to different network platforms without serious impact on the system. This three-tier mechanism ensures *independence* between client

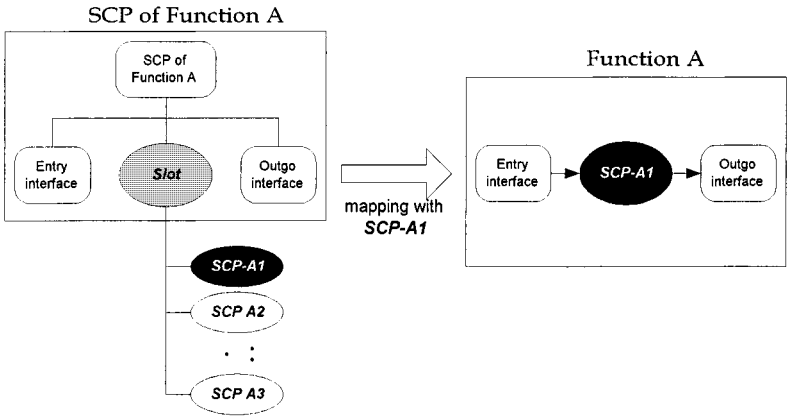


Figure 5. The SCP class hierarchy and the component dynamic packing.

and service components, and supports maintainability and flexibility. The CS-agent also provides the capability for dynamic packing of the service components by mapping alternative configurable parts from the repository to the clients at run-time.

The *substitutable component-parts library* (SCP-lib) is a component-based object repository that organizes function/data objects in a class hierarchy. The SCP-lib differentiates itself from other object libraries by providing both components and their corresponding refinement relations and refined parts so that an identical interface for alternative services can be offered. All of the corresponding configurable components serving the same function are grouped into the same function class so that they can be substituted for each other at run-time. We call this feature the *substitutable component-parts* (SCP) mechanism. Figure 5 shows the detailed structure of the SCP. As an illustration, the SCP mechanism can be used to support the dynamic linking/packing services during the APM roll-back optimization processing, when service requests may be changed at run-time.

The implementation of the ADPCL is based on well-known design patterns. The behavioral patterns like *mediator, strategy, adapter* and *state* patterns are applied to implement the APM. The CS-agent adopts creational patterns like *builder* and *factory method* patterns. The SCP-lib is composed of a class hierarchy and a set of reusable components that are classified and attached with their corresponding classes. Some details are discussed in the following subsections.

3.4. Re-implementation of the PPGE into patterns

3.4.1. Mapping analysis

The process of reengineering the PPGE is a complicated task. We encountered several problems, including the understanding/feature-discovery of the PPGE, the mapping/selection of the right patterns to use, and reengineering the PPGE to a desirable architecture.

The understanding of the PPGE includes the process of decomposing, restructuring and classifying software components of the legacy PPGE. The understanding of the PPGE was mainly based on the conceptual model and documented high-level components. The mapping between patterns and the PPGE behaviour of each subsystem was based on mapping the applicable description of patterns in [2] to the appropriate behaviour characteristics extracted from the PPGE.

On the other hand, ADPCL clearly defined the characteristics and functionality in each layer. We tried to establish mapping relationships in the layers of APM, CS-agent and SCP-lib with the corresponding candidate patterns. This mapping is currently accomplished manually. Table I shows the mapping result from our experiment in this case study.

3.4.2. Re-implementation of the APM portion of the PPGE

The typical process of writing parallel programs in the PPGE is similar to a transformational process. An FEG is transformed phase by phase, like a *transaction*, until the desired optimal MPI code can be generated. The implementation and design of the current PPGE has tightly coupled a set of components with heavy interactions among themselves. For example, the GUI components were mixed with the components of *mesh generator, partitioner/mapper*, Solver, etc. The *mediator* pattern was applied to reconstruct the APM. As shown in Figure 6, the APM contains a GUI mediator and a behaviour mediator, taking care of the GUI component interactions and the PPGE behaviour respectively. To set up a clear boundary between communication tasks, processes and service components, three frameworks are implemented—the communication framework, the state transition framework and the application specific framework.

The *strategy* pattern is used in the *communication framework* since different parallel and distributed models of the PPGE may be used, which usually require different communication strategies. Based on the user's selection of different algorithms in each phase during the run-time, the *state* pattern is used to achieve the run-time configuration and to keep track of the status of each phase. In the application specific framework, the *command*, *adapter* and *strategy* patterns are used. The *command* pattern supports the undo capability during the transaction (process) as shown in Figure 2. The *adapter* pattern is applied to adapt the algorithms with unmatched interfaces. The PPGE adopts many algorithms, which were developed by many other researchers. These algorithms may require different interfaces with different data formats. The *adapter* pattern can bridge the differences between different algorithms. The *strategy* pattern is used to construct a family of alternative algorithms in each phase of parallel programming in the PPGE. For some low-level implementations, the *chain of responsibility* pattern is also used, and some examples are given in the next section. The descriptions of the applicability of the selected patterns can be found in [2].

3.4.3. Re-implementation of CS-agent and SCP-lib of PPGE

While the APM models the interaction and the process of high-level components, the CS-agent coordinates the services of low-level components. The most noteworthy patterns used in the CS-agent are the *factory* and the *builder* patterns. The *factory* pattern helps define a uniform interface for algorithms in each phase of the PPGE parallel programming process and lets subclasses decide which classes to instantiate. The *builder* pattern is used to separate the construction of a complex object like

Table I. The mapping analysis of functional features and candidate patterns in reengineering.

Functional/behavioural features	Characteristic descriptions	Candidate patterns	Implementations of ADPCL
GUI interacting design	The complicated interactions should be eliminated with the isolation between GUI and the processing	mediator, builder, observer	APM
Event switching and controlling	To lower the coupling of events and the corresponding handlers	chain of responsibility	APM
State control	To control and monitor the states of transitions of processes and the whole system	state	APM
Function chain	To lower the coupling within a complex system which consists of a set of sequential subfunctions	mediator, command	APM
Function hierarchy	To lower the coupling within a system which consists of a set of structural subfunctions	mediator, façade	APM
Process-to-process	Interactions between processes	mediator	APM
Process-to-interface	An accordant interface of processes	template	APM, CS-agent
Communication	To provide different communication requests, with a unique interface	mediator, strategy, template	APM, CS-agent
Dynamic binding	Run-time re-configuration	mediator, chain of responsibility	CS-agent
Agent services	Hide the detail of implementation and location of objects from the clients	decorator, adapter, proxy	CS-agent
Process-to-data	To provide a gateway and interface for remote/local data accessing	strategy, proxy, bridge	CS-agent
SCP and component management	SCP dynamic packing of components	Composite, factory method, builder	CS-agent, SCP-lib
Family or class of algorithms	Requests of run-time reconfigurable objects	builder, factory method	CS-agent, SCP-lib
Component wrapping	Provides an identical access manner to all the individual components	command	SCP-lib

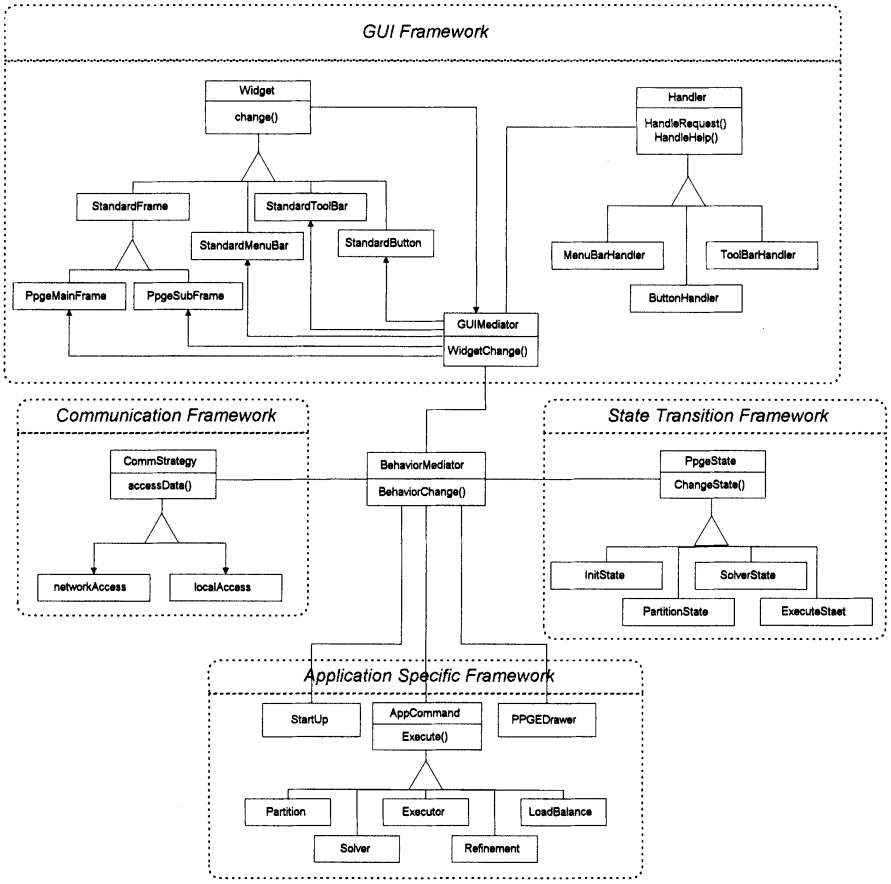


Figure 6. The PPGE APM frameworks in UML notation.

partitioner, solver, etc., as shown in Figure 7, such that the components and their assembly become independent.

The CS-agent also provides the registration entries to the SCP-lib for service index and listing. When adding new SCP classes or SCP parts into the library, or discarding some inappropriate parts, the library sends messages to update the registration of the content on the CS-agent. The CS-agent provides the transparency of dynamic re-configuration from the lower layer to the upper layer APM. The CS-agent increases the independence between the APM and the SCP-lib of the PPGE.

The lowest layer of the ADPCL architecture is the *substitutable component-parts library* (SCP-lib). Serving as a general component repository, the SCP-lib offers the configurable components for the

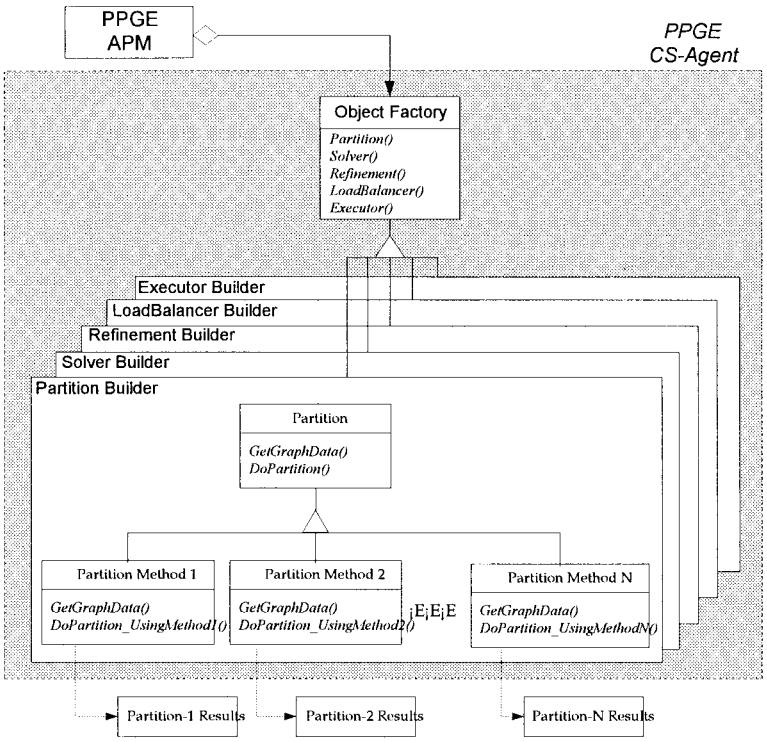


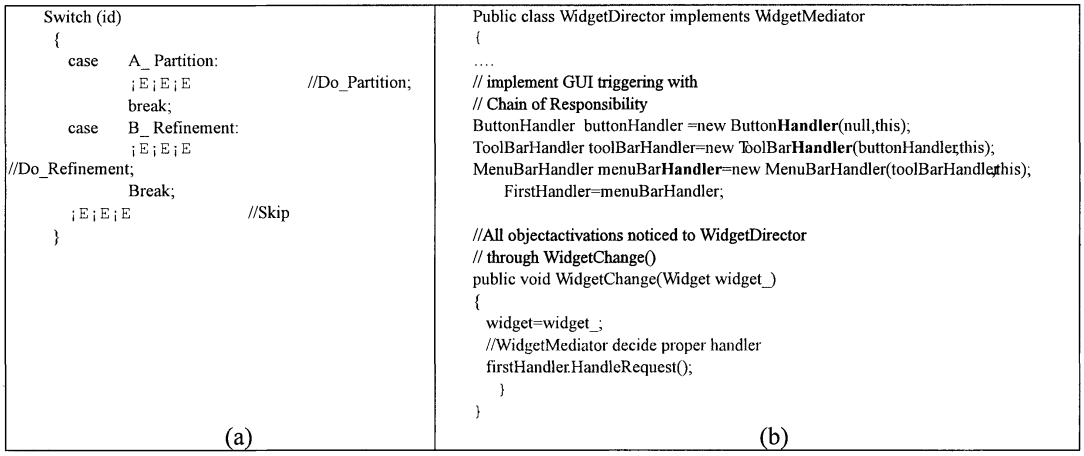
Figure 7. The PPGE CS-agent system model.

PPGE. However, it organizes these components into a set of family components, where members in the same family can be substituted for each other during run-time.

4. EXPERIENCES AND EVALUATIONS

The PPGE was originally implemented in C on a UNIX system, which did not support interoperability and run-time configuration. The target language of this reengineering is JAVA, which has a run-time configuration capability and cross-platform interoperability. Besides, the underlying OO technologies in JAVA make the implementation of our concept and design much simpler.

In this experiment, the understanding of the PPGE and DPs is the essential step for the reengineering process. Fortunately, Gamma *et al.* [2] have offered a well-organized guidebook to DPs; otherwise we might have to spend tremendous amounts of effort to identify and use DPs. The performance of this mapping process is largely dependent on the familiarity of design patterns. We have found that the process becomes much more efficient as the process continues. We believe that if these patterns have



```
Switch (id)
 {
   case   A_ Partition:
          ¡E¡E¡E                    //Do_Partition;
          break;
   case   B_ Refinement:
          ¡E¡E¡E
//Do_Refinement;
          Break;
    ¡E¡E¡E                  //Skip
 }
```

(a)

```
Public class WidgetDirector implements WidgetMediator
{
....
// implement GUI triggering with
// Chain of Responsibility
ButtonHandler  buttonHandler =new ButtonHandler(null,this);
ToolBarHandler toolBarHandler=new ToolBarHandler(buttonHandler,this);
MenuBarHandler menuBarHandler=new MenuBarHandler(toolBarHandler,this);
     FirstHandler=menuBarHandler;

//All objectactivations noticed to WidgetDirector
// through WidgetChange()
public void WidgetChange(Widget widget_)
{
  widget=widget_;
  //WidgetMediator decide proper handler
  firstHandler.HandleRequest();
    }
}
```

(b)

Figure 8. On the left is an original program segment in C for the selection of partition methods; on the right is the same functionality re-implemented as a button handler DP in JAVA.

been taught in classes according to a well-defined architecture, e.g., ADPCL, the reengineering process can be very efficient and cost effective.

The most difficult and time-consuming task in our experiment was the actual reengineering of code to a new target language, in particular the restructuring of old and 'bad' structures in the PPGE into DPs. However, the process can be gradually speeded up when more and more components have been reengineered. This also indicates that a well-structured system is much easier to reengineer. From this case study, we have found that for a long lasting system that requires continual maintenance, the one-time reengineering cost can be easily recovered from reduced future maintenance activities. Besides, the adaptation of new functionality into a pattern-based system is much easier. For example, adding web accessibility to the PPGE was very difficult before reengineering, but it becomes very easy after reengineering because the implementation of new features is done by integrating reusable components.

Figure 8 shows an example of the source code reengineering. In the old PPGE, many segments of source code were like the left side of Figure 8. Functions were triggered by users with a series of `switch cases` testing. All the options were fixed in this segment, and any change to functions would make future modification very difficult. The right side of Figure 8 shows the revision of the same functionality. By using the *chain of responsibility* and the *mediator* patterns, every option activated from the user is detected by the mediator and processed by the corresponding handler. The component-based design implies that any change to the interface (the widgets) or the behaviour (the handlers) has minimized interference with each other.

To compare the results before and after reengineering, the size and functionality of software components have been collected and analyzed. The components of the PPGE have been classified into two categories, the *environment components* (ECs) and the *algorithm components* (ACs). The ECs

Table II. The sizes of the environment components of the original PPGE.

Function description	Size (LOC)	File name
a1. Basic user interface	488	FEGTcmdw.cpp
a2. Flow control	202	FEGTcmdw.cpp
a3. File I/O	93	FEGTcmdw.cpp
a4. Script generation	207	FEGTcmdw.cpp
a5. Component algorithm dialog interface and algorithm executor	463	FEGTmdlg.cpp
a6. Graphic drawer	231	FEGTcnv.cpp
a7. Environment setup	86	FEGTmdlg.cpp

are the components whose functionality is related to the program environment of the PPGE, as shown in Table II, while the ACs are the components that are related to algorithms being used in the PPGE, as shown in Table III. The original total size of ACs of the PPGE is about 27 200 lines of code (LOC) and the size of the ECs is about 1800 LOC.

After reengineering the PPGE to target ADPCL, the ECs of the PPGE were reengineered to the APM and the CS-agent components. The coupling complexity between the APM and the CS-agent became very low, which is a very good indication of reduced future maintenance of the PPGE. The SCP-lib has been reengineered from the ACs of the PPGE. Tables IV, V and VI show the details of all the modules in the APM, the CS-agent and the SCP-lib respectively, including their functional descriptions, sizes in terms of the lines of code, the adopted design patterns, and the degrees of reuse levels.

The degree of reuse levels is an estimated value of how much code and design can be reused during re-implementation. The reused level of a component shows zero if it is newly developed. The total size of the APM is about 2550 LOCs and its average reused level is about 30% while the size of the CS-agent layer is about 1300 LOCs with 0% of reused level. The result reasonably matches our anticipation of the effort of reengineering because the APM re-designs the portions of flow-control and graphical-interface. On the other hand, most of the modules in the CS-agent must be implemented from scratch because there were no agent components in the original PPGE, and therefore the reused level is zero.

On the contrary, the SCP-lib layer adopts most of the algorithm components from the PPGE without any change. In order to integrate with the JAVA environment, *wrapper*, which provides an identical access interface to all the individual components and management components, was developed so that a seamless integration of C++ and JAVA could be achieved, and dynamic-packing and run-time reconfiguration became feasible. Although some new components have been developed, its average reused level is 98%.

Although the migration from the PPGE to the pattern-based PPGE involved extra cost due to the development of new components, we believe that the investment will be paid back from the continuing maintenance. In this case study, we have used one master's student and four undergraduate students in manually reengineering the PPGE. The total person-hours for each phase is as follows: 24 hours for the PPGE understanding, which does not include the understanding of the algorithm modules in the

Table III. The sizes of the algorithm components of the original PPGE.

Algorithm category and algorithm name	Size (LOC)∗	Program name
c1. Refinement		
1. BiSection	1578	1. heap_bi.c
2. Regular		2. heap_reg.c
c2. Partition		
1. AE/ORB	8832	1. aeorb_b_twa.c
2. AE/MC		2. aemc_b_twa.c
3. Parallel H/V		3. hv.c
4. Jostle-MS		4. jostle.sun
5. MLKP		5. k_b_twa.c
6. NNM		6. nnm.c
7. ORB		7. n2orb_remap.c
8.Party		8. party
c3. Load Balancing		
1. BTPLB	8961	1. btplb.c
2. BINOTPLB		2. binotplb.c
3. CBTPLB		3. cbtplb.c
4. CWA		4. cwa.c
5. MWA		5. best_new_mwa.c
6. PCMPLB		6. pcm.c
7. TWA		7. twa.c
8. DD		8. dd.c
c4. Solver		
1. Euler	7896	1. euler.c
2. Laplace		2. par_laplace.c
3. Linear Elasticity		3. lin_elasticity.c
4. Poisson		4. poisson.c
5. Conjugate Gradient		5. con_gradient.c
6. Gauess-Seidel		6. gauess.c
7. Jocobi Iterative		7. jocobi_iterative.c
8. Multigrid		8. multigrid.c

PPGE, 220 hours for target system analysis and design, and 140 hours for the re-implementation of the new PPGE. The reengineering process took less than two months of effort for the PPGE with the size around 30 000 LOCs. The data also show that the correct design documents, e.g., published papers or technical reports, can indeed save a lot of time for system understanding.

To evaluate the improvement of scalability of the PPGE after reengineering, we compared the efforts of adding a new component into both the old PPGE and the new PPGE systems. As shown in Table VII, adding a new component into the old PPGE required modification of components of a1, a2, and a5 with an impact degree of 65%, where the impact degree is the total number of changed modules divided by

Table IV. The description of the APM components.

Function description	Size (LOC)	Design patterns adopted	Reuse level
A1. Static user interface	320	mediator	0
A2. Flow control	583	mediator state chain of responsibility	40%
A4. Environment setup	112	mediator	30%
A2. Parser (for parsing to parameter definition of new components)	553	iterator	0
A4. Network manager	140	bridge	0
A6. Dynamic user interface	546	builder observer	0
A9. Graphic drawer	300		80%

Table V. The description of the CS-agent components.

Function description	Size (LOC)	Design patterns adopted	Reuse level
B1. Component loader	184	strategy proxy	0
B4. Dynamic binding	288	mediator chain of responsibility	0
B7. SCP and component manager	642	composite factory method builder	0
B11. Network access	144	bridge	0
B13. File import/export	50		0

Table VI. The description of SCP-lib components.

Function description	Size (LOC)	Design patterns adopted	Reuse level
C14. Wrapper and SCP management components	335	command	0
C1–C4 (the same as c1–c4 in Table III)	27 266		100%

the total number of modules. The new pattern-based PPGE has an impact degree of 0%, since the CS-agent can integrate this new component through dynamic binding without affecting other components.

To evaluate the adaptability of both the old and new PPGE, we enhanced the PPGE to become an Internet-accessible system. Table VIII shows an encouraging outcome. The only modification needed for the reengineered PPGE was the addition of a tiny network-access class, which can be fitted into the PPGE using the *strategy* pattern. The impact degree for the new PPGE was less than 1%, but 44% for the original PPGE. The difference is obvious and substantial.

We are confident of the extensibility and reconfiguration capability of the reengineered system. Obviously, with the assistance of the reengineered PPGE, parallel programming is much easier. Now it

Table VII. The impact analysis for system scalability: adding a new component.

Versions	Modules need to be changed	Impact degree (no. of changed modules/ no. of total modules)
Original PPGE	a1. Basic user interface a2. Flow control a5. Component algorithm dialog interface and algorithm executor	65% $(a1 + a2 + a5)/$ $(a1 + a2 + a3 + a4 + a5 + a6 + a7)$
New PPGE	NIL	0%

Table VIII. The impact analysis for system adaptability: Network ability extension.

Versions	Modules need to be changed	Impact degree
Original PPGE	a1. Basic user interface a2. Flow control a3. File I/O	44% $(a1 + a2 + a3)/$ $(a1 + a2 + a3 + a4 + a5 + a6 + a7)$
New PPGE	Add a new network-access class (30LOCs)	$<1\%$

is easy to explore various possible algorithms and methods, to pursue improved PPGE run performance, and to extend the coverage of the solutions (that historically consumed tremendously laborious hours of work). Adding new algorithms in each phase is now feasible and can be done easily without affecting the other subsystems. We ascribe this positive contribution with the reengineering result and automatic tool support to the FEM process, which makes parallel programming much easier and cost-effective.

As a result, the PPGE is easily reconfigured to generate execution results and detailed comparisons on more different algorithms and with the assignment of different numbers of processors, which has been a painful task for the designers in a traditional environment. With dynamic binding, programmers are encouraged to strive for optimal solutions and to get better results.

5. CONCLUSIONS

Instead of redesigning a legacy system, reengineering is more cost-effective and less risky to the enterprise. In this paper, we propose the ADPCL architecture and use design patterns as the basis for reengineering legacy software systems.

The ADPCL is a three-tier architecture that sets up a clear boundary between the application processes and the data resources to achieve low coupling relationships. The ADPCL can be seen as the integration of MVC [34] and CORBA [35]. MVC (model/view/controller) addresses the advantage

of independency between interfaces, processes and data models. CORBA (common object request broker architecture) is a three-tier architecture that establishes a standard for client–server remote-accessing services. The ADPCL provides a higher level of abstraction than MVC by adopting the characteristics of design patterns. Compared with CORBA, our architecture contains the features of the active component agent and run-time reconfiguration capability. These features greatly promote the maintainability and extensibility of reengineered systems.

The ADPCL architecture and the experimental reengineering case study have achieved the following goals:

- The ADPCL exhibits the features of high reusability, easy evolution and a design-pattern based architecture.
- The ADPCL isolates the application process and the data so that the components have low coupling and low dependency among themselves. From this case study, the reengineered PPGE shows an encouraging improvement on system scalability and adaptability.

The features of dynamic binding and run-time reconfiguration can reduce the cost of maintenance tremendously. The design of the CS-agent is especially suitable for incorporating new components. In this experiment we added Internet accessibility to the PPGE.

This case study indicates that the pattern-based reengineering approach is feasible. The same idea can be applied to other software systems. In the future, we would like to work on the automation of design pattern discovery, retrieving and mapping, all of which were done manually in this case study. We would like to generalize the ADPCL to cover more legacy systems in different application domains.

ACKNOWLEDGEMENTS

The work was supported in part by the National Science Council of Taiwan under the contract NSC89–2213–E–029–004. The authors thank Yu-Ping Kuo of Feng Chia University, and Hsuan-Yu Chu, Wen-Da Lian, Cho-Min Chou, Shiow-Chwen Chiou and Chia-Wei Chang of TungHai University for their assistance in implementing this reengineering project. The authors also thank the Journal's anonymous reviewers for their helpful suggestions.

REFERENCES

1. Booch G. 1994. *Object-Oriented Analysis and Design with Applications*; Benjamin-Cummings Publishing Co.: Redwood City CA; 3–26.
2. Gamma E, Helm R, Johnson R, Vlissides J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Publishing Co.: Reading MA; 12–86.
3. Coad P, North D, Mayfield M. 1995. *Object Models–Strategies, Patterns & Applications*; Prentice-Hall International: Englewood Cliffs NJ; 18–67.
4. Larman C. 1997. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*; Prentice-Hall International: Englewood Cliffs NJ; 391–424.
5. Chu CW, Lu CW, Chang CH, Liao CJ, Chung YC. 1999. A parallel program generation environment for solving PDEs on distributed memory computing environments. In *Proceedings of the SEKE'99*; Knowledge Systems Institute: Skokie IL; 267–272.
6. Shewchuk JR, O'Hallaron DR. 1998. *Archimedes: A System for Unstructured PDE Problems on Parallel Computers*; Carnegie Mellon University: Pittsburgh PA; 83–97. Also at URL http://www.cs.cmu.edu/afs/cs.cmu.edu/project/quake/public/www/archimedes.html [1 December 1999].

7. Aykanat C, Ozgüner F, Martin S, Doraivelu S. 1987. Parallelization of a finite element application program on a hypercube multiprocessor. In *Proceedings of the SIAM Second Conference on Hypercube Multiprocessors*; Society for Industrial and Applied Mathematics: Philadelphia PA; 662–673.
8. Aykanat C, Ozgüner F, Ercal F, Sadayaooan P. 1988. Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. *IEEE Transactions on Computers* **37**(12):1554–1568.
9. Garey MR, Johnson DS. 1979. *Computers and Intractability, A Guide to Theory of NP-Completeness*; W. H. Freeman and Co.: San Francisco CA; 17–44.
10. Bank RE, Sherman AH, Weiser A. 1983. Refinement algorithms and data structures for regular local mesh refinement. In *Transactions of the Tenth IMACS World Congress*, Stepleman R *et al.* (eds). *Scientific Computing*, Volume 1. IMACS/North-Holland Publishing Company: Amsterdam, Netherlands; 3–17.
11. Baden SB. 1991. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM Journal on Scientific and Statistical Computing* **12**(1):145–157.
12. Bänsch E. 1991. An adaptive finite-element strategy for the three-dimensional time-dependent Navier–Stokes equations. *Journal of Computational and Applied Mathematics* **36**(1):3–28.
13. Pilkington JR, Baden SB. 1996. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems* **7**(3):288–300.
14. Simon HD. 1991. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering* **2**(2/3):135–148.
15. Preis R, Diekmann R. 1996. *The PARTY Partitioning Library User Guide*; Department of Computer Science. University of Paderborn: Paderborn, Germany; 31–68.
16. Williams RD. 1990. *DIME: Distributed Irregular Mesh Environment*; *Technical Report C3P-861*, California Institute of Technology: Pasadena CA; 86–107.
17. Bank RE. 1994. *PLTMG: a software package for solving elliptic partial differential equations, Users' Guide 7.0*; SIAM Publications: Philadelphia PA; 22–56.
18. Ramaswamy S, Sapatnekar S, Banerjee P. 1997. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems* **8**(11):1099–1115.
19. Liu P, Wu J. 1998. Supporting efficient tree structures for distributed scientific computation. *Journal of Information Science and Engineering* **14**(1):79–105.
20. Shaw M, Garlan D. 1996. *Software Architecture: Perspectives on an Emerging Discipline*; Prentice-Hall International: Englewood Cliffs NJ; 39–75.
21. Wirfs-Brock RJ, Johnson RE. 1990. Surveying current research in object-oriented design. *Communications of the ACM* **33**(9):105–124.
22. Rine DC. 1997. Supporting reuse with object technology. *IEEE Computer* **30**(10):43–45.
23. Sommerville I. 1995. *Software Engineering*, (5th edn.); Addison-Wesley Publishing Co.: Reading MA; 700–712.
24. Arnold RS (ed.). 1994. *Software Reengineering*; IEEE Computer Society Press: Los Alamitos CA; 25–72.
25. Pleszkoch MG, Linger RC, Hevner AR. 1992. Eliminating non-transferable paths from structured programs. In *Proceedings Conference on Software Maintenance*; Los Alamitos CA: IEEE Computer Society Press; 156–164.
26. Gall H, Klosch R, Kofler E, Wurfl L. 1994. Balancing in reverse engineering and in object-oriented systems engineering to improve reusability and maintainability. In *Proceedings COMPSAC '94*; IEEE Computer Society Press: Los Alamitos CA; 35–42.
27. Gall H, Kolsch R, Mittermeir R. 1996. Application patterns in re-engineering: identifying and using reusable concepts. In *Proceedings of the 6th International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems (IPMU'96) vol. III*; Special session on Software Reusability; Granada Spain, 1099–1106.
28. Gleich U, Kohler T. 1997. Tool-support for reengineering of object-oriented systems: position paper on the FAMOOS-project. In *Proceedings of ESEC/ACM FSE/WOOR'97*; Technical University of Vienna, Information Systems Institute: Wien, Austria, 53–61.
29. Wang Y, King G, Court I, Ross M, Staples G. 1997. On built-in tests in object-oriented reengineering. In *Proceedings of ESEC/ACM FSE/WOOR'97*; Technical University of Vienna, Information Systems Institute: Wien, Austria, 54–65.
30. Jahnke J, Zundorf A. 1997. *Rewriting poor design patterns by good design patterns. Technical Report TUV-1841-97-10*, Technical University of Vienna Information Systems Institute, Technical University of Vienna; Wien, Austria, 43–79.
31. Bridges P, Doss N, Gropp W, Karrels E, Lusk E, Skjellum A. 1995. *User's Guide to MPICH, a Portable Implementation of MPI*; at URL http://csep.hpcc.nectec.or.th/mpi/mpiuserguide/paper.html.
32. Burns G, Daoud R, Vaigl J. 1994. *LAM: An Open Cluster Environment for MPI*; Ohio Supercomputer Center, Colombus Ohio.
33. Meyer J. 1994. *Message-passing interface for Microsoft Windows 3.1. Master's Thesis*, Department of Computer Science, University of Nebraska at Omaha, Omaha NE; 11–42.
34. Krasner GE, Pope ST. 1988. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* **1**(3):26–49.

35. Vinoski S. 1997. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine* **35**(2):46–55. {February}.

AUTHORS' BIOGRAPHIES

William C. Chu is an Associate Professor in the Department of Computer and Information Science at the TungHai University, Taiwan. From 1994 to 1998, he was an Associate Professor at the Department of Information Engineering at the Feng Chia University, Taiwan. Prior to that, he was a research scientist at the Software Technology Center of the Palo Alto Research Laboratories of Lockheed Missiles and Space Company, Inc., where he received special contribution awards from Lockheed in both 1992 and 1993. In 1992, he was a Visiting Scholar in the Department of Engineering Economic Systems at Stanford University, where he was involved in projects related to intelligent knowledge-based expert systems. His current research interests include software reengineering, maintenance, reuse, software quality and e-commerce. William received his M.S. and Ph.D. degrees from Northwestern University Evanston, Illinois, in 1987 and 1989 respectively, both in Computer Science.
E-mail: chu@cis.thu.edu.tw

Chih-Wei Lu is currently a Ph.D. student in Computer Engineering at Feng-Chia University, Taiwan. His areas of research interest include component-based software engineering, design patterns, software reuse and software maintenance. Chih-Wei received his B.S. degree in Computer and Information Science from TungHai University in 1987 and his M.S. degree in computer science from the University of Southern California in 1992.
E-mail: cwlu@plum.iecs.fcu.edu.tw

Chih-Peng Shiu is a Master's student in the Department of Information Engineering at Feng Chia University, Taiwan. His research interests include object-oriented programming, web-based technologies and software engineering. E-mail: jpshiu@soft.iecs.fcu.edu.tw

Xudong He is at Florida International University in Miami, Florida, working in the areas of formal methods and software testing techniques. He served as an associate professor at North Dakota State University at Fargo in North Dakota from 1989 to 1999. Xudong received his B.S. and M.S. degrees in Computer Science from Nanjing University, China in 1982 and 1984 respectively, and his Ph.D. degree in Computer Science from Virginia Polytechnic Institute and State University (Virginia Tech), Virginia in 1989. E-mail: hex@cs.fiu.edu

